Spring 4-1-1995

Addressing the Scalability Problem in Visual Programming ; CU-CS-768-95

Wayne V. Citrin
University of Colorado Boulder

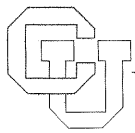
Richard S. Hall
University of Colorado Boulder

Benjamin G. Zorn
University of Colorado Boulder

Addressing the Scalability Program in Visual Programming

Wayne Citrin
Richard Hall
Benjamin Zorn

CU-CS-768-95



University of Colorado at Boulder
DEPARTMENT OF COMPUTER SCIENCE

Addressing the Scalability Problem in Visual Programming

Wayne Citrin

Richard Hall

Benjamin Zorn

CU-CS-768-95 April 1995

University of Colorado at Boulder

Technical Report CU-CS-768-95
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309

Addressing the Scalability Problem in Visual Programming

Wayne Citrin
Department of Electrical and Computer Engineering
Campus Box 425
University of Colorado
Boulder, CO 80309-0425, USA
Tel: 1-303-492-1688
E-mail: citrin@cs.colorado.edu

Richard Hall, Benjamin Zorn
Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430, USA
Tel: 1-303-493-4398
E-mail: rickhall@cs.colorado.edu
zorn@cs.colorado.edu

ABSTRACT

Visual programming languages have often been criticized for their lack of scalability, particularly in the way in which they become unusable when used to describe large programs. This lack of scalability manifests itself in three ways: complex and unreadable diagrams, viscosity of representation, and excessive use of screen real estate. We describe a way in which language design and programming environment design may be integrated to address this problem. The VIPR [Visual Imperative PRogramming] language and its associated environment address this problem through the use of potentially unlimited nesting, fixed bounds on the screen real estate occupied by procedures regardless of their complexity, and the use of perspective in the language paradigm, along with the use of zooming and animation in the programming environment. Since VIPR is an ongoing project, the paper also describes ways in which we plan to address scalability issues in the future.

KEYWORDS

Visual languages, graphical editors, programming environments

INTRODUCTION

One of the main criticisms leveled against visual programming languages concerns their lack of scalability. This lack of scalability may take three forms. First, in visual languages based on graphs (that is, with nodes and edges), as programs or program fragments become more complex, the diagrams become hopelessly complex and unreadable. This increase in complexity seems to occur whether the language is based on a data-flow or a control-flow model. The problem stems from the existence of edges in the diagram, and the fact that the edges may connect widely separated nodes, leading to arbitrary graph complexity. Numerous measures have been proposed for this complexity as it affects readability, including edge density, node density, edge crossing density, and edge angular resolution [6].

A second, related, scalability problem concerns what Green refers to as *viscosity* [8, 9]. Viscosity refers to the resistance of a representation to local changes. For example, as nodes are added to a graph-based visual language, it may eventually become necessary to rearrange the entire graph in order to maintain readability, the alternative being overly crowded or otherwise complex diagrams. Because of this, most visual languages are referred to as being highly viscous, unlike less viscous textual representations. Green shows that viscosity is related to other usability issues, including the necessity of planning ahead when laying out a program.

The third problem related to scalability is the reputed lower density of visual representations, manifested in greater screen real estate requirements for visual programs. Although many investigators seem to accept this claim without question, comparative studies indicate that programs in certain visual languages may actually be more compact than equivalent textual programs. (The representation of certain programs in R-technology, a Russian visual language [14], is more compact than the equivalent Pascal or LabView [12] programs, for example.) In many, if not most, cases, however, experience shows that visual programs do take up more space.

The VIPR [Visual Imperative PRogramming] language and its associated programming environment address these issues in a number of ways. Through the use of spatial relations (particularly containment) to indicate semantic relations, VIPR reduces diagram complexity by minimizing the number of explicit edges that must be drawn and viewed. The viscosity problem is addressed by limiting the number and types of transformations the VIPR environment may perform on a program as it is entered. The screen real estate problem is addressed in several ways. First, the use of spatial containment for semantics means that the size (screen area) of a procedure remains fixed, regardless of how complex the procedure grows. Second, the use of containment may lead to interpretation of the programs in an apparent third dimension that may be used to handle the growing complexity. Third, programs may be arbitrarily shrunk while retaining a great deal of recognizable detail. Fourth, the environment employs zooming in the editor in order to view and modify deeply nested constructs, and it employs animation in the execution in order to trace execution into those constructs.

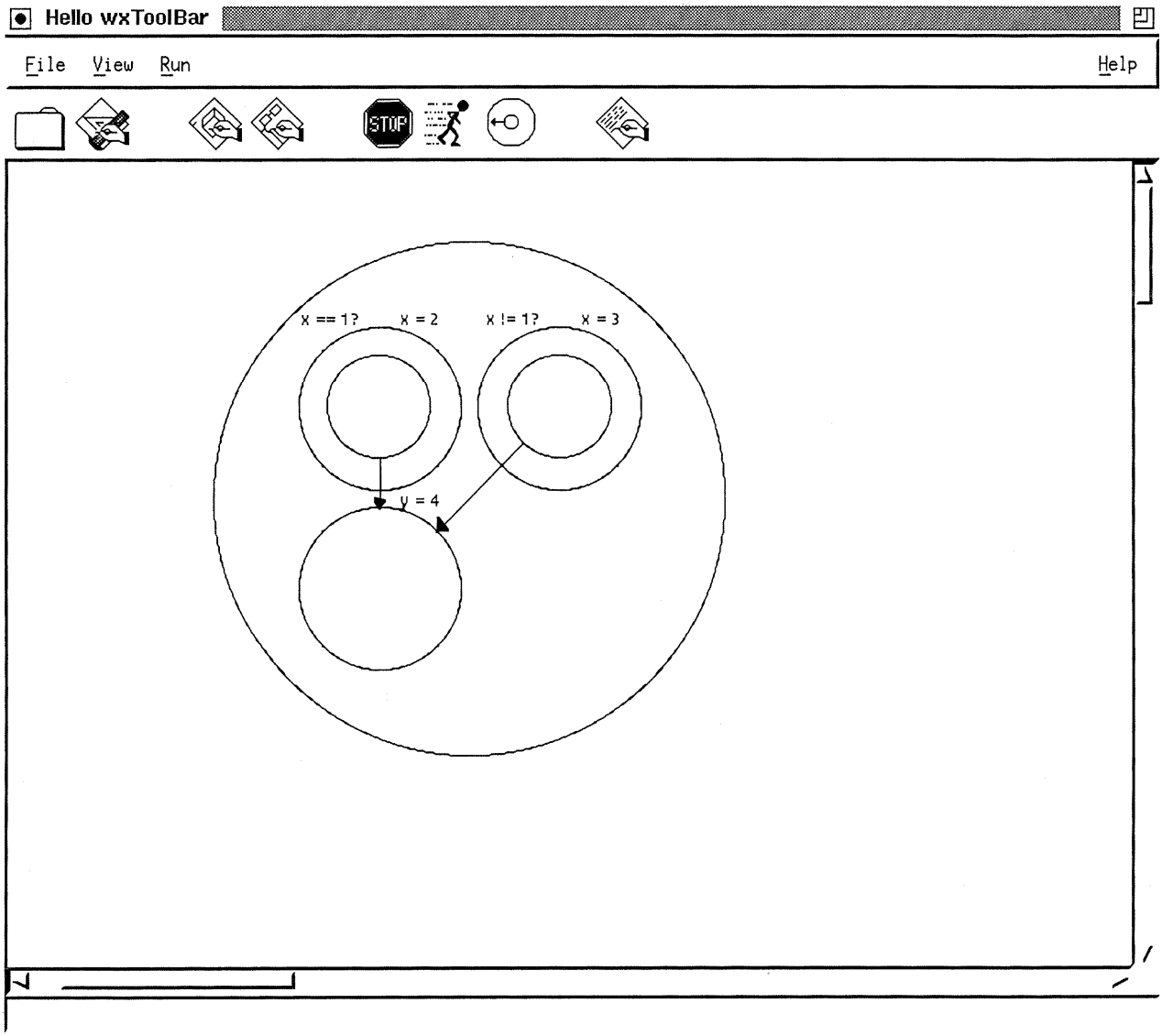


Figure 1. VIPR example

VIPR OVERVIEW

VIPR is a completely visual imperative programming language based on the model provided by Pictorial Janus [11]. By "completely visual," we refer to those languages in which the static program and current program state (that is, both the static and dynamic aspects of the program) are incorporated in the same diagram, and the language semantics are specified by a set of allowable graphical transformations on the current program configuration. In the latter respect, completely visual languages resemble the graphical transformation languages [1, 7, 15]. Kahn [10, 11] conjectures that the simpler and more transparent semantics of completely visual languages make programs easier to write.

Detailed discussion of the semantics of VIPR is beyond the scope of this paper; see [2] for such a discussion. In brief, a VIPR program (figure 1) consists of a set of nested rings, one of which is distinguished as the *state ring*. The state ring carries program state (particularly, the values of variables), and the other rings carry guards and actions. Execution occurs at the boundary of the state ring and the outermost ring within the state ring; if that outermost ring's guard is true, that ring *merges* with the state ring and disappears, and the state is altered according to the action associated with the merged ring. Selection rules control execution when more than one outermost ring is a candidate for execution, and substitution mechanisms, denoted by arrows, allow the normal flow of control to be altered. Using this small set of simple mechanisms, VIPR can model conventional conditionals, case statements, iterators, procedure calls, and parameter passing.

Figure 1 shows a simple VIPR program displayed in the VIPR programming environment. The outer ring is the state ring, and is assumed to carry the program state. (Note that

the environment shown in this paper is a preliminary version – subsequent versions will include a graphically distinguished state ring, including explicit representation of current variable values.) Inside the state ring are three instruction rings (two of which have further contents) and a pair of substitution arrows.

Of the three outermost instruction rings, the two upper rings are considered candidates for execution, since the third, lower, ring is the target of substitution arrow (see below) and execution of such target arrows is suppressed.

Each candidate ring has a guard. The guards are tested, and one ring whose guard evaluates to true is selected as the next instruction to execute. All other candidate rings and their contents are eliminated. Assuming that the current value of X is 1, the upper left ring is selected for execution, and the other candidate ring is eliminated. This stage in the computation is shown in figure 2a.

At this point, the candidate ring executes by merging with the state ring, and by altering the state carried by the state ring, in this case by setting the value of X to 2. The candidate ring itself vanishes and its contents now become the new candidate rings, as is shown in figure 2b.

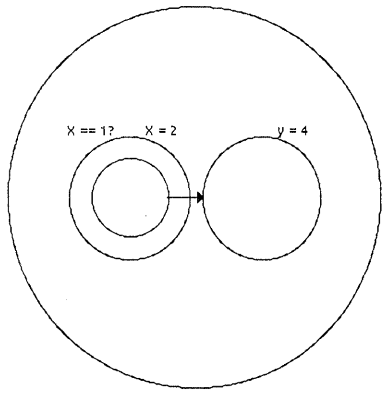
The current candidate ring has no guard and no action, but has an arrow exiting it. This arrow indicates that the construct at the destination of the arrow may be substituted for the ring at the source. Thus, the right-hand ring is substituted for the left-hand ring, as is shown in figure 2c. Substitution arrows are used to mimic the effects of gotos, procedure calls, and any other non-sequential constructs.

Finally, the remaining statement ring executes by merging with the state ring, and the state is altered by setting y to 4. At this point, the state ring is empty and execution halts. It can be seen that the construct shown in figures 1 and 2 models the if-then-else construct.

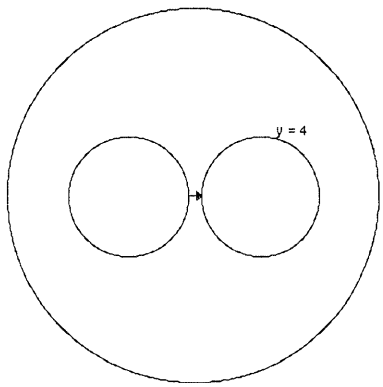
Through the explanation above, it can be seen that nesting of rings is used to denote normal sequencing – a given ring executes before the rings nested inside it. Although the other mechanisms are significant in the language (in particular the substitution arrows shown in figure 1), they are not relevant to the treatment of VIPR scalability and will not be further discussed in this paper.

Although figure 2 shows snapshots of an executing VIPR program, the graphical transitions representing computation steps are smoothly animated. A number of sample animations of VIPR programs are available on the World Wide Web at http://soglio.colorado.edu/Web/vipr.html.

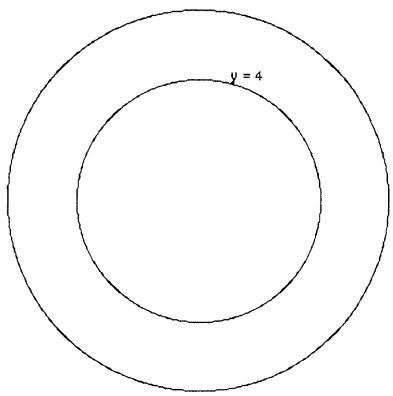
Although VIPR constructs map closely onto conventional textual program constructs, VIPR differs significantly from other such visual constructs, such as flow charts and Nassi-Shneiderman diagrams [16]. Most significantly, VIPR programs possess a completely visual semantics, but equally important, from the point of view of scalability, VIPR programs express sequencing through a spatial containment relation, while the other representations employ connectedness or adjacency. The next section shows that such differences are extremely significant.



(a)



(b)



(c)

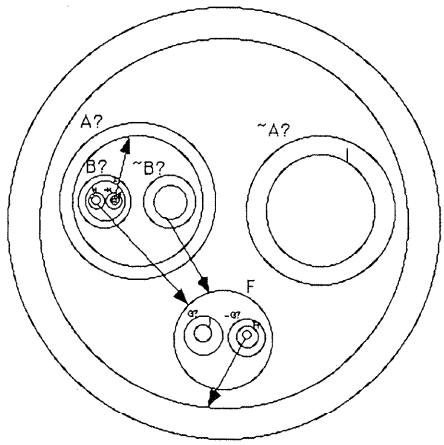
Figure 2. Execution of a VIPR program. (a) After application of selection rule. (b) After application of sequencing rule. (c) After application of substitution rule

VIPR AND GRAPH COMPLEXITY

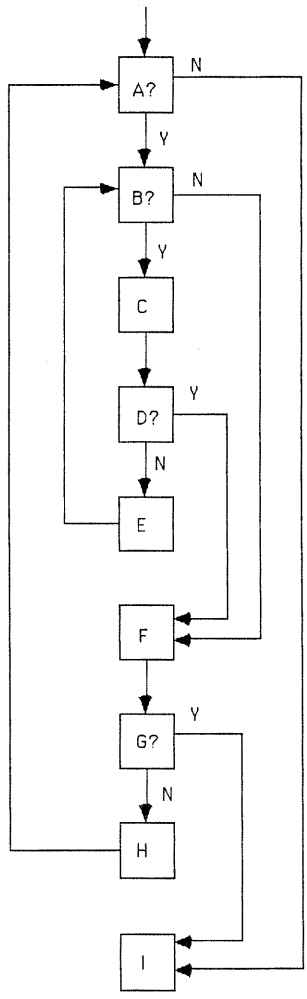
Research on graph complexity and its relation to drawing and readability has identified a number of factors influencing the readability of a graph [6]. These factors include minimization of the number of edge crossings and minimization of the graph area. Graph complexity is a problem in most graph-based visual languages because all data and control flow are represented by graph edges, and visual programs quickly become complex. A typical solution, employed by popular visual languages like Prograph [5] and LabView [12], is to divide the graphs into multiple pages or windows, but requiring multiple windows decreases usability, particularly on small screens.

VIPR addresses this problem by eliminating the need for edges to specify normal sequential control flow and selection. Thus, most edges that would appear in a control-flow visual program in a graph-based language need not appear in a VIPR program. Only those constructs requiring other types of control flow than simple sequencing (such as iterators, procedure calls, and the joining of two branches of a conditional) need to employ edges. Thus, VIPR programs can be compressed substantially while maintaining their readability, and larger programs may be displayed on a single screen using VIPR than using other visual languages. Figure 3 illustrates this property. Figure 3b shows a program of two nested loops with additional loop exits in a hypothetical graph-based control flow visual language. Figure 3a shows the equivalent VIPR program. Figure 3c shows the equivalent textual program. The graph-based program contains 9 nodes and 13 edges. The equivalent VIPR program contains 21 rings and only 4 edges. Since readability of graphs is generally concerned with the number of edges and the complexity of their crossings, the VIPR program remains more readable as it is compressed than the graph-based program, and the VIPR program, as presented, clearly occupies less space than the graph-based program, while maintaining readability. As graph-based programs become more complex, graph layout becomes more complex in a way that it does not in VIPR, since problems of node placement and edge routing must be addressed. In VIPR, the fact that containment is the primary spatial relation means that layout considerations are much simpler. Finally, the disks described by nested VIPR rings share screen area, and therefore the space occupied by the VIPR program does not increase as the number of rings increases in the way that the space occupied by a graph-based program would increase as the number of nodes increased, since the graph nodes may not overlap.

It should be noted that the number of edges in the VIPR program in figure 3a may be further reduced to 2 by collapsing copies of the ring F and its contents into the rings at the source of the arrows whose destination is F. Thus, edges may be removed at the cost of adding rings. There are times when such transformations may allow further compression of the program, and future versions of the VIPR environment may support such transformations.



(a)



(b)

Figure 3. (a) VIPR program. (b) Equivalent control-flow graph program. (c) Equivalent textual program (on following page)

```
while A do begin
   while B do begin
         C;
         if D then break;
         E;
         end;
   F;
   if G then break;
   H;
   end;
I;
```

(c)

Figure 3. (c) Equivalent textual program

VIPR AND LANGUAGE VISCOSITY

Green [8] has identified viscosity as the resistance of a language or environment to small local changes. For example, in inserting a new node into a control-flow graph, it may be necessary to rearrange other nodes and edges in order to avoid creating overly complex graphs. To accommodate this viscosity, users often spend a great deal of time planning ahead the entry of program elements.

As an example of viscosity in visual languages, consider the program in figure 3b. Suppose that the user wishes to insert a new node to be executed between nodes D and E. This would be impossible without either globally rearranging the graph to provide more space between nodes D and E, or by placing the new node outside the existing graph and running edges to and from the new node, thereby increasing the number of edges and probably increasing the number of edge crossings. The latter solution is not desirable from the standpoint of graph complexity, and the former solution is complex to perform by hand. In theory, an editor for this language could perform this global rearrangement automatically, but such transformations are complex, particularly when the nodes of the program are not linearly arranged, as they are in figure 3b. Such complex transformations are generally not supported in graphical editors.

In contrast to the above example, textual languages and their editors are generally considered low-viscosity. Text editors take care of addition and deletion of statements automatically; the main examples of viscosity in textual languages concern the management of bracketing constructs (**begin-end**, for example), and indentation.

The fact that most control-flow relations need not be represented explicitly with edges allows VIPR to address the viscosity problem. The ways in which rings may be added to a VIPR program are severely constrained. One may add a ring to the end of a sequence, in which case the VIPR environment places the ring at the inside of a sequence of nested rings (figure 4a). One may insert a ring between two nested rings, in which case the VIPR environment compresses the inner of the two rings (and its contents) sufficiently to allow the new ring to be inserted (figure 4b). One may add a sibling ring to set of alternatives. In this case, the existing sibling rings are rearranged and they and their contends are automatically compressed to yield sufficient room for the new ring (figure 4c). For the reasons discussed in the previous section, compression of VIPR constructs may be accomplished to a large extent without sacrificing comprehensibility, and where comprehensibility is finally lost, the zooming facility of the VIPR editor allows the compressed constructs to be easily navigated. It is very important to note that these are the *only* graphical transformations that need be supported by the VIPR editor for local modifications.

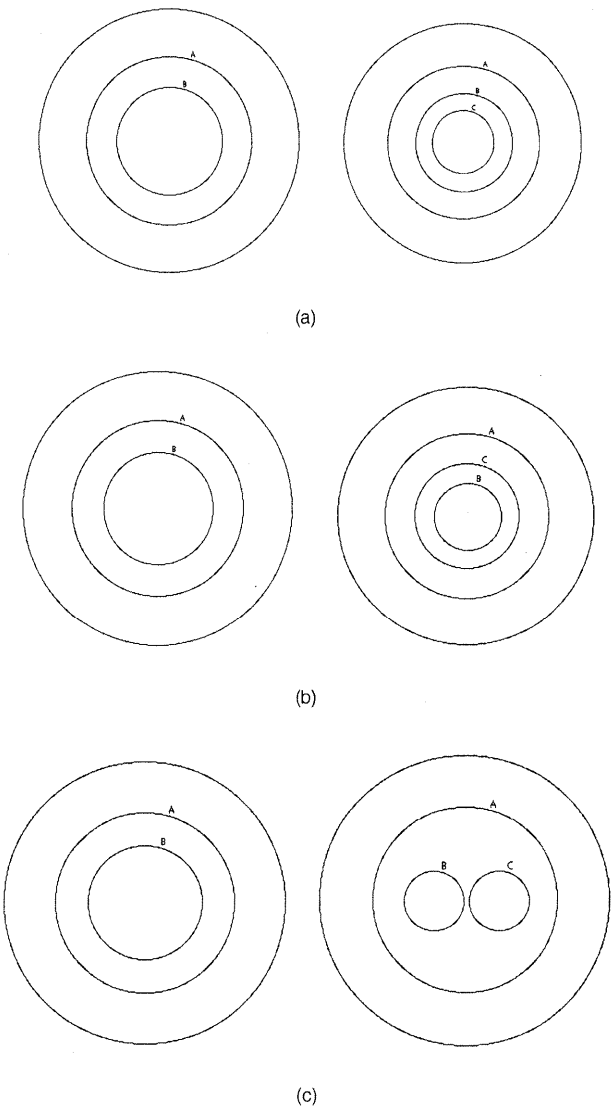


Figure 4. (a) Adding an innermost ring. (b) Inserting a ring. (c) Adding a sibling ring

The other way in which the user may add a ring to a VIPR program is as an external procedure. In this case, the user may draw the ring anywhere on the drawing surface, and the normal viscosity problems reappear. However, the program

aggregates being dealt with are much larger than individual statements and there are fewer such items in a typical VIPR program than there are nodes in a conventional control-flow or data-flow graph-based visual program. In addition, when adding rings to the newly created external procedure, the same constraints hold as when adding rings to the main procedure, as described above. Thus, editing and extending existing procedures is non-viscous; only adding new procedures and making procedure calls exhibits viscosity. This is a substantial advance over other graph-based visual languages.

We are currently addressing viscosity issues involving external procedures. This effort is discussed in the future work section.

VIPR AND SCREEN AREA

It is often claimed that visual programs occupy more screen space than equivalent textual programs. VIPR addresses this problem in several ways.

First, the fact that spatial containment is used to denote sequencing means that the size of a VIPR procedure is bounded, no matter how complex the procedure becomes. Assuming that the size of the outermost ring is fixed, new rings are added in progressively smaller sizes in the interior. Although the rings may become quite small, the use of zooming allows such small rings to be read and navigated, as described below. Figure 5 shows how a simple procedure and a complex procedure developed from it occupy the same amount of space.

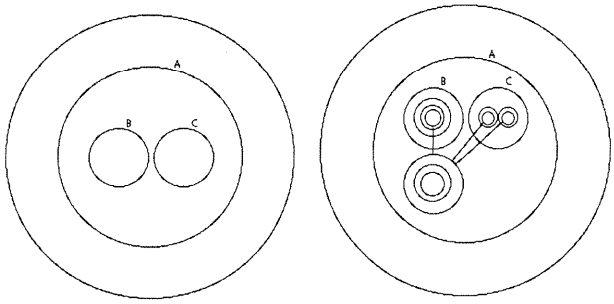


Figure 5. Bounded size, independent of complexity

A second way in which the design of VIPR accommodates the screen real estate problem is to expand programs into a virtual third dimension. Although two-dimensional nesting is the main topological relation used to relate rings, this nesting relation may be interpreted as a perspective drawing in a third dimension. In such an interpretation, a sequence of nested rings may be viewed as a perspective view down a tunnel (figure 6). In such an interpretation, zooming may be interpreted as movement down that tunnel. Similarly, executing programs may be visualized through travel down that perspective tunnel. Evidence suggests that users are willing to accept smaller and more cluttered representations if they may be interpreted as due to receding perspective in a third dimension and if ways are provided to navigate in that third dimension [13].

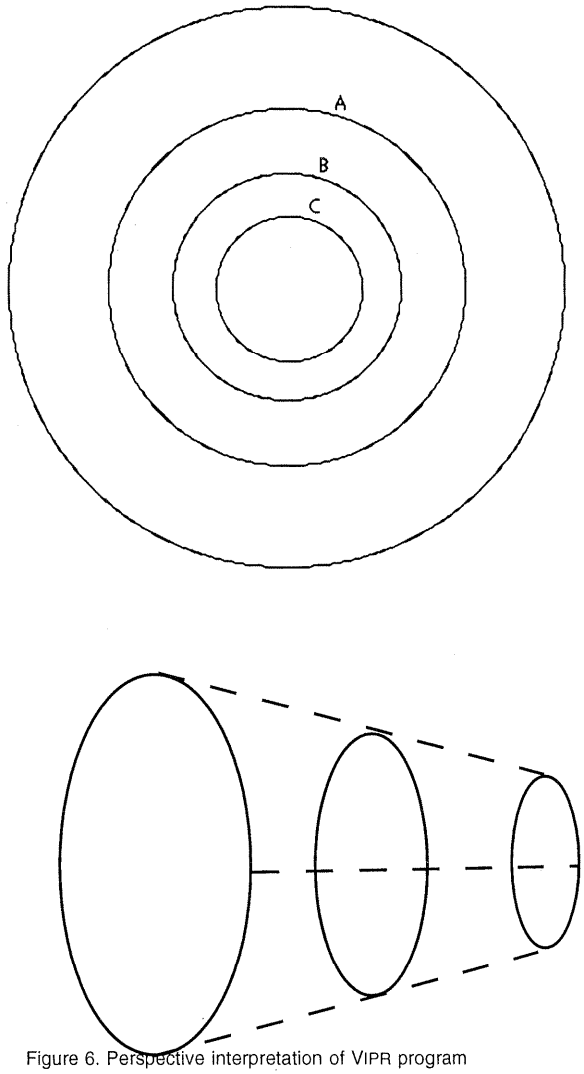


Figure 6. Perspective interpretation of VIPR program

Third, programs may be arbitrarily shrunk while retaining a great deal of recognizable detail. We have already seen how VIPR procedures may be compressed with only a limited loss in readability, due to the lack of edges needed to express sequencing relationships. It can also be shown that entire programs may be compressed without complete loss of information. The form of a program, particularly its call graph, is still visible at high compression. It is possible to recognize individual procedures and observe their calling relationships. This sort of recognition is more difficult when conventional graph-based visual languages are used, due to the fact that connectedness denotes sequentiality and it is therefore not necessary for statements of the same procedure to actually be close to each other, and it is impossible with textual languages because detail and readability are lost as the text shrinks.

Figure 7 gives an example of how large-scale details may emerge when a VIPR program is compressed. At a glance, we can see that the program consists of three procedures. The two topmost procedures contain conditional constructs with two alternative cases each, while the bottom procedure contains a case construct with three cases. The number of transfers of control, their sources and destinations, are all immediately apparent, even at a high degree of reduction in program size.

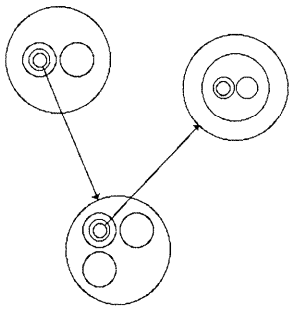


Figure 7. Observation of gestalt in a VIPR program

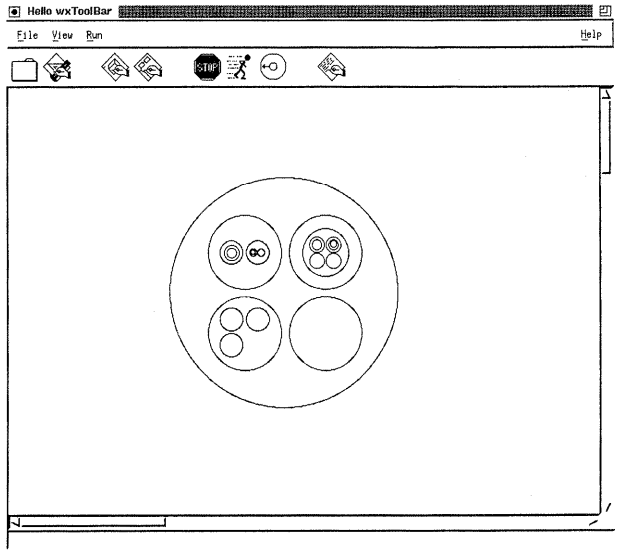
Finally, the VIPR environment supports zooming in order to view and modify deeply nested constructs, and it employs animation in the execution in order to trace execution into those constructs. These features allow highly compressed detail to be recovered in editing and execution, respectively. When a complex VIPR procedure is created, so that statement rings are deeply nested and small, it is easy to zoom into the procedure in order to magnify the nested rings, observe detail, and make modifications.

Figure 8 shows how the VIPR zooming facility appears to a user. The top portion of the figure shows a complex VIPR program in its full-program view, displaying the large structure of the program. In the bottom portion of the figure, the view has zoomed in on the upper left-hand quadrant of the program, recovering detail that was difficult to discern in the full-scale view. The VIPR environment provides facilities for zooming in and out in large and small units, as well as the ability to scroll horizontally and vertically through a large or highly magnified diagram.

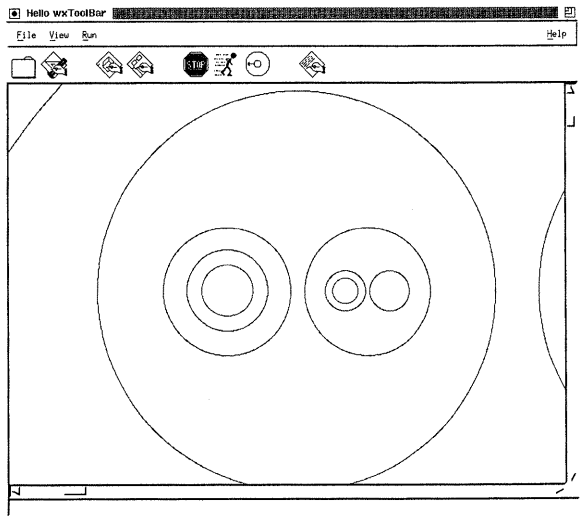
Similarly, when observing execution, the VIPR environment displays the merging of the outermost state ring with the state ring, and animates the enlargement of the executing procedure's interior. This animated zooming allows detail of small rings to grow into visibility as the corresponding statement comes closer to execution.

CONCLUSIONS AND FUTURE WORK

The VIPR project shows how the design of a visual language and its programming environment may be integrated to effectively address problems of visual language scalability. The use of spatial containment, or nesting, as the indicator of sequencing allows reduction in the complexity of representations and allows topological transformations, particularly compression, that reduce screen space preserving readability to a large extent, something that does not occur in graph-based visual languages based on a node-and-edge model. When readability is lost through excessive compression, it can be easily recovered through the zooming support provided by the VIPR environment.



(a)



(b)

Figure 8. (a) A deeply nested procedure. (b) Zoomed view of the interior

In addition to compression, the constraints on where rings may be added to a procedure and the simple automatic transformations that are performed when the rings are edited address the problem of viscosity in visual languages. This solution allows VIPR procedures to remain editable whereas conventional graph-based visual language procedures lose editability as they become more complex.

The features of VIPR described in this paper are generally concerned with scalability within single program unit. We are currently attempting to address scalability problems involving multiple program units using VIPR's ability to preserve readability even when a program is highly compressed. One promising approach involves 'fisheye' views of large programs, where the program unit which is the focus of attention is enlarged, and other units are compressed. The programming environment will support smoothly animated transitions between alternate fisheye views. Figure 9 shows three fisheye views of the program in figure 7, each focusing on a different program unit. Such fisheye transformations also address the problem of viscosity in programs with multiple procedures.

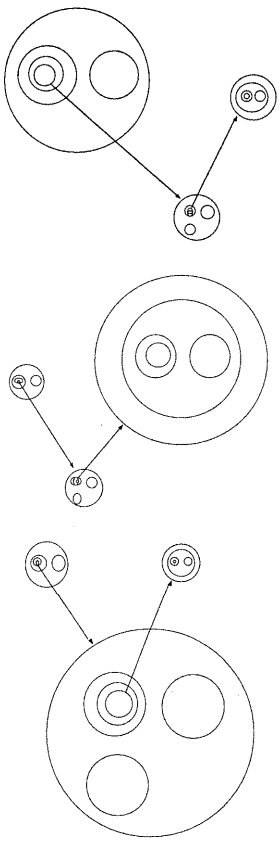


Figure 9. Use of fisheyeing to navigate multi-unit programs

We are also exploring more sophisticated circle packing methods. The ring layout algorithm in the current VIPR environment is rather simple-minded, and, although relatively simple to implement, leaves a large amount of unusable and unused space in a VIPR construct. More efficient circle packing algorithms will yield denser programs and will allow more complex program constructs to occupy a given amount of space.

The VIPR programming environment is implemented in C++ and Tcl and employs the wxwin windowing environment on a SPARC5 workstation. Work is currently being done to extend the VIPR language to incorporate visual expressions [4] and object-oriented constructions [3]. In addition, we are attempting to improve the usability of the zooming and navigation facility, as well as the quality of the animation.

ACKNOWLEDGMENTS

The work presented here was supported by the Colorado Advanced Software Institute (CASI) and USWEST Technologies. The authors thank Scott Wolff of USWEST Technologies for his advice and support.

REFERENCES

[1] Bell, B. and C. Lewis, "ChemTrains: A Language for Creating Behaving Pictures," in *IEEE Symposium on Visual Languages*. 1993. Bergen, Norway, 188-195.

[2] Citrin, W., M. Doherty, and B. Zorn, "Formal Semantics of Control in a Completely Visual Programming Language," in *IEEE Symposium on Visual Languages*. 1994. St. Louis, 208-215.

[3] Citrin, W., M. Doherty, and B. Zorn, "The Design of a Completely Visual OOP Language," in *Visual Object-Oriented Programming,* Burnett, M., A. Goldberg, and T. Lewis, eds. 1995, Prentice-Hall: Englewood Cliffs, NJ. 67-93.

[4] Citrin, W., R. Hall, and B. Zorn, "A Visual Lambda Calculus," Technical report CU-CS-757-94, Department of Computer Science, University of Colorado, Boulder, January 1995.

[5] Cox, P. T., F. R. Giles, and T. Pietrzykowski, "Prograph: a step towards liberating programming from textual conditioning," in *Proc. 1989 IEEE Workshop on Visual Languages*. 1989. Rome, 150-156.

[6] Esposito, C., "Graph Graphics: Theory and Practice." *Comput. Math. Applic.*, 1988. **15**(4): 247-253.

[7] Furnas, G. W., "New graphical reasoning models for understanding graphical interfaces," in *Human Factors in Computer Systems: CHI '91 Conference Proceedings*. 1991. New Orleans, 71-78.

[8] Green, T. R. G., "Programming Languages as Information Structures," in *Psychology of Programming,* Hoc, J.-M., *et al.*, eds. 1990, Academic Press: New York.

[9] Green, T. R. G. and M. Petre, "When Visual Programs Are Harder to Read then Textual Programs," in *Human-Computer Interaction: Tasks and Organization, Proc. ECCE-6 (6th European*

Conference on Cognitive Ergonomics), van der Veer, C. G., *et al.*, eds. 1992, CUD: Rome.

[10] Kahn, K. M., "Towards Visual Concurrent Constraint Programming," Technical Report SSL-91-092, Xerox Palo Alto Research Center.

[11] Kahn, K. M. and V. A. Saraswat, "Complete Visualizations of Concurrent Programs and Their Executions," in *IEEE Workshop on Visual Languages*. 1990. Skokie, IL, 7-15.

[12] Kosodsky, J., J. MacCrisken, and G. Rymar, "Visual Programming Using Structured Data Flow," in *IEEE-CS Workshop on Visual Languages*. 1991. Kobe, Japan, 34-39.

[13] Mackinlay, J. D., G. G. Robertson, and S. K. Card, "The Perspective Wall: Detail and Context Smoothly Integrated," in *Proc. Conf. on Human Factors in Computing Systems (CHI '91)*. 1991. New Orleans, 173-179.

[14] McHenry, W. K., "R-technology: a Soviet visual programming language." *Journal of Visual Languages and Computing*, 1990. **1**(2): 199-212.

[15] McIntyre, D. W. and E. P. Glinert, "Visual Tools for Creating Iconic Programming Environments," in *IEEE Workshop on Visual Languages*. 1992. Seattle, 162-168.

[16] Nassi, I. and B. Shneiderman, "Flowchart Techniques for Structured Programming." *SIGPLAN Notices*, August 1973. **8**(8).